Proof planning and Maintainability of AI Systems

Helen Lowe

DAI Research Paper No. 663

Proof planning and Maintainability of AI Systems *

H. Lowe
Department of Artificial Intelligence
University of Edinburgh, 80 South Bridge
Edinburgh, EH1 1HN
Telephone 44-31-650-2728
Email hcl@uk.ac.ed.aisb

November 15, 1993

Abstract

Configuring large computer hardware systems is a complex task and efforts to automate it have spawned knowledge based systems which have proved difficult to maintain. We propose a new approach which breaks away from the traditional rule-based IKBS paradigm and have implemented a prototype system in Prolog in which to test out our ideas. In this paper we give a logical formalism for configuration and use a *proofs as configurations* approach; this approach is analogous to program synthesis *via* proofs as programs and equally rigorous. Meta-level control is achieved through *proof plans*, which embody common strategies for guiding inference. As well as ensuring soundness, in that only *legal* configurations can be synthesized, the system also facilitates maintenance because of its clean separation of object-level knowledge, heuristics, and control strategies. Using the hardware configuration domain by way of example, we have provided an important proof of concept in the quest for reliable, robust, and maintainable knowledge based systems.

*Thanks are due to the SERC and Hewlett Packard Research Laboratories, Bristol, who have supported me for three years with a SERC/CASE award; Alan Bundy, Frank van Harmelen, David Basin, and Geraint Wiggins of the Department of Artificial Intelligence, Edinburgh; and Richard Hull, Stefek Zaba, and many others at HP Labs.

1 Introduction

Generally speaking, expert systems, supposedly based on the distilled knowledge of experts, are *not reliable* (Bundy, 1987b). Many knowledge based systems suffer from the difficulty of separating object-level and meta-level knowledge. Knowledge bases are compiled with the aid of human experts who are often unable to unravel the two, or even understand the distinction. As new objects are added or new rules discovered it becomes increasingly difficult to maintain the system. Trying to hone systems so that they are *efficient*, by the use of control information, or to incorporate *design* considerations in more open-ended problems, only exacerbates the difficulties in trying to ensure the resulting system is reliable, robust, and maintainable.

Bundy (1987b) proposed a technique known as *proof planning*, which was originally developed for use in theorem proving and program synthesis, for IKBS in general. This technique makes use of methods written in a meta-logic to facilitate the task of writing proof plans to guide the search for proofs in automatic theorem proving, and possesses the following properties:

1. Efficiency, because the combinatorial explosion is avoided, or at least greatly mitigated.
2. Generality, because a proof plan may be applicable to many proofs.
3. Maintainability, because the separation of factual knowledge from control knowledge means that either may be changed without affecting the other.
4. Explanatory power, because control decisions can be explained at the appropriate level, rather than by generating long chains of low-level choice points in the inference process.

These properties are important for *any* knowledge based system. Thus proof plans can provide a useful vehicle for expressing strategies for problem-solving in other domains, including non-mathematical ones.

As a proof of concept, we looked at the problem of how to synthesize a configuration which meets a a specification for a computer system. This synthesis should result in a term representing all the components needed, together with details of the connections between them. We shall examine how computer hardware can be synthesized to meet a given specification, using a technique analogous to program synthesis (see Section 4). The careful separation of *object-level knowledge* from *meta-level control* and *heuristic knowledge* is an important feature of our approach (Section 5). This has the benefit of facilitating maintenance, in a domain which traditionally has been bedevilled by maintenance overheads. Although other systems, notably XCON, have moved in this direction (Rosenbloom *et al*, 1985; Soloway *et al*, 1987; Mumford & Macdonald, 1989; Bachant & Soloway, 1989; Barker & O'Connor, 1989), in our configuration system this separation is both strict and explicit. We shall show how object-level knowledge may be extracted and formally represented in such a way as to allow the utilization of techniques analogous to program synthesis, to perform tasks such as synthesizing computer configurations which meet specifications. Such an approach makes the task of maintaining knowledge bases more tractable and reliable.

It should be stressed that this was no easy task: knowledge bases have to be compiled with the aid of human experts who are often unable to unravel the different kinds of knowledge, or even realize the distinction. However, if this is not done, then, as new objects are added or new rules discovered, it becomes increasingly difficult to maintain the system. Thus the initial effort in unravelling these different kinds of knowledge, which is forced upon us using our approach, should in practice prove worthwhile in the long run.

2 Other approaches

There are many configuration systems around: every major manufacturer of hardware has one and it is an interesting domain for the general researcher to test ideas on. Automated configuration has a long history in AI terms: one of the earliest, most successful knowledge based systems was built to tackle the problem of configuration (McDermott, 1982). The motivation was simply good business sense for a task which was difficult, and where mistakes were frequently made, but which seemed essentially a matter of remembering and following rules: a prime candidate for automation. The knowledge of the experts was canned and made available to the inexpert and ignorant.

In this comparative study, we shall concentrate on rule-based systems, since although constraint-based systems and other paradigms exist (Frayman & Mittal, 1987), these have been for hardware systems on a much smaller scale. Since the problem of configuring large modern computer systems is inherently underconstrained, it is unlikely that these would be successful for computer systems of the scale tackled by XCON. Other approaches, *e.g.* the semantic net approach of of Searls & Norton (1990), were also tried on small systems and seem unlikely to scale up.

XCON (Bachant & Soloway, 1989; Barker & O'Connor, 1989) is an customer order *checker* — it takes an order for a configuration (*i.e.* a list of components) and attempts to configure them, adding any parts which are necessary but which have been omitted from the order, and making other corrections. XSEL works at a higher level, taking account of customer requirements to assemble an order for hardware components, and is interactive, in that the user is prompted for components; the system cannot generate them in any way. At the end of the interactive stage of the process XSEL passes an order to XCON.

XCON has a reputation for being difficult to maintain. Circumstantial evidence is that in the published accounts (Barker & O'Connor, 1989) the problem of training personnel to work on XCON is prominent; the solution proposed is to send all new staff on an AI course on entry.

The main problem is that one change can often have a large knock-on effect in terms of how many rules need modification. There is no separation of domain knowledge ("facts") and control knowledge ("strategy"). Most of the conditions in the condition-action rules define the context in which the rule is to be considered. Hence control information is implicit rather than explicit, and mixed in with the factual knowledge. One piece of knowledge may be contained in many rules, so that if this piece of knowledge needs to be modified, then so must all these rules — and they have first to be identified. This is difficult for anyone but an experienced maintenance programmer — experienced, that is, in XCON itself. "As the size of the knowledge base increases, it becomes even more difficult to determine all of the interrelationships between the rules" (Freeman, 1985). By 1987 XCON had 6,200 rules, a tall order for maintenance. However, it is seen as a successful system, Harmon (1987) citing that 99% of its 50,000 orders in 1986 were properly specified.

Attempts have been made to improve the control aspect of XCON, as described in Soloway *et al.* (1987), by using a higher level language, namely RIME, where the relationship of RIME to OPS5 (which XCON is written in) is analogous to that between FORTRAN and assembly code. The main claim made for XCON-*in*-RIME is that control knowledge is made explicit: both in terms of the problem-solving method (claimed to be domain-independent); and the entering of different problem spaces or *contexts* (said to be domain-specific).

However, constant updating of XCON and XSEL is a feature of rapidly changing product lines, and, as described in Barker & O'Connor (1989), specialized training of personnel is essential to carry out this task. This reflects the fact that there is insufficient separation of control knowledge from factual and heuristic knowledge. The task of maintaining the system is not straightforward, needing specialized knowledge of the computer system as a whole, and an understanding of AI techniques. Personnel responsible for product updating cannot perform their task in isolation; likewise neither can configuration engineers responsible for maintaining performance heuristics and control strategies.

All these systems suffer to some degree in that:

- Maintenance is difficult because facts, heuristics, control knowledge, and context information are inextricably intermixed.
- The rule-based system was not based on a firm logical foundation.
- Revolutionary changes in technology pass them by. Automated systems for configuring more up-to-date hardware are fundamentally different from their predecessors. There is no reason to suppose that they will not become obsolete in their turn.

The solution proposed to the maintenance of XCON, namely training new personnel in AI techniques (Barker & O'Connor, 1989), only serves to highlight the problem!

An intricate knowledge of expert systems should not be needed for routine maintenance. A knowledge based system should be capable of being maintained by people who did not necessarily have knowledge *of the system as a whole*. In other words, someone whose job it was to maintain the price and product list as new components came into production and obsolete ones were deleted would not have to know anything other than how to maintain the particular files which hold the information relevant to this task. Similarly, an engineer should be able to maintain the part of the system requiring her particular expertise independently of the rest of the system.

3 Motivation for our approach

3.1 General advantages of logical formalism

Configuration is a task involving much attention to detail and some search problems. For example, by getting it wrong a company can lose a great deal of money and goodwill. It is easy to make mistakes, *e.g.* by forgetting vital cables or not realizing the most efficient configuration.

A common business practice is for a company (we shall refer to this as the *client company*) to issue a specification and to invite tenders for a configuration. The tendering companies each produce descriptions of candidate configurations meeting this specification, together with details of capabilities and costs. The client company will not necessarily accept the lowest tender (in terms of absolute cost) but may aim to accept the tender which proffers the best "value for money". (This is a consequence of the fact that the specification may be no more than a minimal requirement.)

Once the tender is accepted, a contract is signed between the client and the successful tenderer. This contract will state that the computer firm agrees to provide a system based on the proposed configuration which at least meets the given specification for the agreed cost. This means that, if the configuration turns out to be deficient, the computer firm must make good the shortfall and bear any extra cost involved. In many ways, the loss of goodwill, from the attendant delays in providing the promised service is, in the long term, even more expensive for the firm involved.

Since it is so important, one might wonder why mistakes occur with such frequency. The answer seems to be a combination of rapidly changing product lines, and much low-level detail which is difficult to absorb. Often the computer firm's representative is an expert in sales at the higher level of specification, but cannot be expected to keep track of the low-level configuration details.

In more precise terms, the problem is in the generation of *unsound* "solutions", *i.e.* the objects which are generated are not *legal* objects (configurations). They do not obey the laws of configuration, *e.g.* that every device in the configuration should have a legal connection.

We have developed an axiomization of configuration theory which allows the use of theorem proving techniques. This renders a more precise definition of the problem and enables us to reason about it in a rigorous way so as to ensure soundness.

Given the problem of rapidly-changing product lines, a prime goal is for the system to be readily maintainable. It will be seen that the particular formalism we have chosen, with its separation of knowledge and control, facilitates the maintenance of *all* types of knowledge in the system.

3.2 Isolating well-formedness considerations

Many of the rules concerning configuration can be thought of as constraints. These may be divided into two categories:

1. "Well-formedness", *i.e.* what kinds of objects can be connected together.
2. Limits, *e.g.* on how many devices can be connected to a given channel.

The first of these can be hived off as well-formedness checking, and embedded in the object-level theory as type-checking. (See Section 6.2 for details.) These rules are absolute, in the sense that they may never be relaxed.

We can represent the connection of two components by the pair $< comp\_x, comp\_y >$. So an operation which semantically corresponds to connecting two components together is checked logically by ensuring that an object of the form $< comp\_x, comp\_y >$

is well-typed. The theory defines which objects may be well-typed. Anything else is not and cannot be synthesized.

In the case of the other category of rules, however, many of these so-called "rules" can be relaxed. What we have is a hard-and-fast version of the rule, which cannot be broken (a *hard* constraint, or *absolute* limit); and a corresponding *soft* constraint, which represents a *heuristic* limit. The hard constraints are part of the object-level theory, whilst the soft constraints are stored as heuristics. For example, suppose we have a rule which says that to guarantee an efficiently running configuration, we should not have more than n_1 devices attached to a channel. In fact, it is physically possible, and often acceptable, to have n_2 devices connected, where $n_2 > n_1$. Then the object-level rule is:

$$\forall x : \mathbf{channel}.\ \forall yl : list(\mathbf{device}).$$

$$(< x, yl >: \mathbf{channel} \times list(\mathbf{device}) \quad \rightarrow \quad length(yl) \leq n_2)$$

and the heuristic version of this replaces n_2 with n_1.

Complex meta-level reasoning is needed in using heuristics (we use the technique of proof plans, as explained in Section 6.7), but the rules governing legal connections are straightforward. There is benefit in separating them out in this way. We separate the problems of

- Finding solutions which are legal. The object-level theory ensures this.
- Finding solutions which are "better" in some sense. Reasoning with heuristics at the meta-level ensures that these are found, if possible.
- Finding solutions (of some kind) efficiently, *i.e.* by guiding the process of inference. The use of strategies encapsulated by proof plans facilitates this.

3.3 Explicit representation of control knowledge

Because the search problem is, in general, quite considerable, *some* kind of control is needed for all but small hardware systems synthesis. We have chosen to encapsulate control strategies explicitly, using a meta-level language to be used in the preconditions slots of methods. The main benefit of separating object-level knowledge from meta-level knowledge in this way is ease of maintaining both kinds of knowledge.

Our main aim was to produce a prototype system which was capable of being maintained by people who did not have knowledge of the whole system. So, for example, some files would hold product information and as such would be updated as new products came on to the market and old ones became obsolete. Others would hold heuristic information and need updating by engineers from time to time as new information became available. Yet another file would hold information designed to encapsulate strategies for configuration: at the present time this holds the knowledge of configuration strategy that we have discovered and implemented so far. It can be constantly updated in the light of experience.

The knowledge is separated in such a way as to render it possible for different people to update files *independently* of other knowledge held by the system. Thus the addition of one new component does not render the whole configuration system obsolete. There are no hidden "knock-on" effects, such as those reported in systems where knowledge, control, and context are inextricably linked.

4 The proofs as programs paradigm

Bates & Constable (1985) give a method for synthesizing algorithms from proofs. If we express the relationship between the input and the output of a program as $spec(input, output)$, then an algorithm alg may be synthesized by finding a *constructive* proof of the theorem

$$\forall input.\exists output.spec(input, output)$$

and from this extracting the algorithm alg such that

$$\forall input.spec(input, alg(output))$$

We use an analogous technique for synthesizing configurations. Suppose we have a specification for a computer configuration, $spec(c)$. We wish to synthesize a hardware configuration c obeying $spec(c)$. For example, the specification might state (translated into informal language) that the configuration should have a certain number of terminals for running particular applications, that it should have at least a certain amount of disk storage, that it should have printers capable of certain speeds, tasks, *etc.* The synthesized c should:

- Obey the explicit terms of the specification — have the correct number and type of terminals, printers, disk drives, *etc.*
- Be a *legal* configuration — function correctly, obeying the general laws for configurations, *i.e.* possess a processor of sufficient power, enough memory, backup devices, *etc.* In addition, all devices must be correctly connected up. These laws may be formalized as a general theory of configuration.

We synthesize such a configuration from the object-level theory by proving the theorem (more properly, the conjecture: we could be given an unrealizable specification)

$$\exists c.spec(c) \tag{1}$$

where c is a well-formed object of type **configuration**, and $spec(c)$ is the specification that c must satisfy.

An alternative way of thinking about (1) is to introduce a meta-variable C and to prove

$$spec(C), \tag{2}$$

where $spec(C)$ is a conjunct of goals expressing the required properties of the configuration.

In proving conjecture (2), C is instantiated to a well-formed term. This is a gradual process; C starts out as a simple meta-variable but acquires some structure early on in the proof, for example

$$C \equiv proc :: L$$

where $proc$ is instantiated but L is not: read this as "C is a processor $proc$ and some other terms". Later in the proof L in turn acquires some structure, and by the end of the proof is fully instantiated.

5 Domain knowledge

5.1 Separation of knowledge

In many tasks we observe that the "knowledge" is of different kinds.

1. Factual:
   - The total capacity of a string of disk drives is equal to the sum of the capacities of each disk drive in the string;
   - $< \textit{this disk drive} >$ has capacity 571 Mb;
   - Every device in a configuration has a means of connection in the configuration.
2. Heuristic:
   - It is best to have fewer than the legal maximum of four disk drives on a channel.
3. Control:
   - When configuring a hardware system from scratch it is best to follow steps in this order: choose processor, then configure disk drives, then assess backup needs,...

Referring to the problems mentioned towards the end of Section 2:

- In order to be expressible in logic, a system needs to be based on such a separation. This separation enables each type of knowledge to be encoded *declaratively* if we so wish. This is of the utmost importance, if we are to be able to check that the logical formalism given accords with our understanding of the semantics. We need to be able to check, separately and independently:
  - That the facts represented are "true", or at least what we intend (*e.g.* the rules of configuration).
  - That procedures are captured correctly.

If these two are inextricably intermixed, the configuration task becomes unacceptably hard.

- A system based on a clean separation is easier to maintain, because knowledge is encoded declaratively.
- It is more likely that we will be able to salvage *something* in the face of technological innovation, provided it is not too extreme. For example, if there is a radical change in storage methods and components, then it will not affect the top-level architecture. Methods of generating partial configurations which are not affected by the changes to affected components will remain unscathed also.

The knowledge used to carry out the task of synthesizing computer configurations is of three kinds:

1. Object level axioms, *i.e.* hard and fast rules, such as "all devices must be connected"; "there must be no more than four disk drives connected to this kind of channel";
2. Heuristic knowledge, *e.g* "for an efficiently running configuration, do not connect more than three disk drives to this kind of channel";
3. Control knowledge, *e.g.* knowledge about the order in which subtasks should be carried out, such as "synthesize devices before card cages";

and that it is important that these three categories of knowledge be kept separate and explicit so that they may be easily maintainable.

It is clear that object level knowledge must be updated as new products come into being and others become obsolete. We described how this changing knowledge is represented and will show how it can be updated.

Heuristic knowledge is, or should be, changing with time and circumstance; for instance, the fact that particular configurations lead to inefficiency may only be learned from experience of actual running configurations.

Explicitly and separately held control knowledge enables us to update the configuration as whole structures or new kinds of products are added or altered, and may mean that the system can be generalized and used for other, similar tasks.

We shall now look at examples of each type of knowledge.

5.2 Object-level knowledge

Suppose we have a component of type **channel**, say *ex_channel*, and associated with this component we have a set of limits:

1. No more than six devices may be connected to it.
2. No more than four objects of type **disk-drive** may be connected to it.

So, for example, we may connect four disk drives and two printers to *ex_channel*, but not five disk drives and one printer, or three disk drives and four printers.

Suppose now that we have a component *ex_cardcage*, of type **card-cage**, and that this component has associated with it the following constraints:

1. No more than four of them in a configuration.
2. No more than four objects of type **channel** to be configured in them.

Each of these four limits are *hard constraints*, in that they may never be relaxed. Any object not conforming to them is not legal.

5.3 Heuristics

Suppose now that it has been discovered that configuring the maximum number of components legally possible in a configuration *may* lead to inefficiently running computer systems. Equally, it may not — it is contingent on circumstances. However, it has been discovered that the application of simple heuristics can *guarantee* efficiently running systems, whatever the circumstances. Suppose in the example above, these heuristics amount to more stringent limits as follows:

1. No more than five devices should be connected to an *ex_channel*.
2. No more than three objects of type **disk-drive** should be connected to an *ex_channel*.

and

1. No more than four *ex_cardcages* in a configuration (no change from the above).
2. No more than three objects of type **channel** to be configured in an *ex_cardcage*.

The changes from the previously given limits represent relaxed, *soft limits*, *i.e.* these limits are desirable for some reason and in some sense, but objects not conforming to them may still be legal.

Sometimes soft compatibility constraints are discovered during the lifetime of a component. For instance, there is a certain model of printer which, when connected to the same channel as a disk drive, causes the computer system to run slowly if both are operational at once. The soft constraint states that this model of printer should not be connected to any channel which has disk drives connected to it. This is an example of the kind of heuristic information which may accumulated over the lifetime of a class of hardware systems. For example, the knowledge that this particular printer caused problems might only be known after feedback from engineers and customers. A new model of printer might come on the market whose performance in certain situations had not been predicted. We can lose these heuristic constraints too — for example, as a result of feedback from customers and engineers, a component may be altered to obviate the problem. It is almost as important that such constraints can be removed when they are no longer appropriate as it is that they are incorporated in the first place.

5.4 Control knowledge

We might put this together into a control strategy which is summarized thus:

1. Try to find a configuration which obeys all heuristic limits.
2. If not possible, try to find a configuration which breaks as few heuristics as possible.

Here, we are treating all heuristics as equally important. However, this leads to us "preferring" breaches of the *channel-per-cardcage* heuristic, since one breach here gives us a "breathing space" while we load devices on to the "extra" channel up to heuristic limits before we have to consider any more breaches. So the strategy can be expanded as follows:

1. Try to find a configuration which obeys all heuristic limits.
2. If not possible, ignore the heuristic:

    > No more than three objects of type **channel** to be configured in an *ex_cardcage*.

    but try to find a configuration which meets both the other heuristics.
3. Otherwise, just try to find a legal configuration.

It should be emphasized that this is only one of a number of possible strategies. Another strategy might be to break one particular heuristic as little as possible, whilst adopting a "don't care" attitude to breaches of other heuristics. Suppose we had 38 disk drives to configure. Rather than allow one breach of the *channel per card cage heuristic*, we might allow two breaches of the *disk drive per channel* heuristic.

Apart from managing heuristics, there are other types of control knowledge. For example, when configuring devices on channels, it pays to configure the most restricted devices first, *i.e.* devices which may not share channels with certain other devices in the configuration. On a more global level, there are advantages to performing the task in a particular order. There may be no universally appropriate order, but, given particular specifications, there may be a way of ordering the various subtasks so as to cut out excessive search. For example, in many cases it is best to decide on the processor first, as this may determine which other devices are permissible. On the other hand, if we were trying to upgrade an existing system by adding (say) ten more terminals and replacing the processor, then we might try to configure existing devices (disk drives, *etc.*) and then infer which processor would best meet the bill of supporting the new devices whilst still being appropriate for the old ones. These, more global, strategies are expressed as proof plans (see Section 6.7).

6 Basic architecture

6.1 Implementation

The main task addressed in designing the architecture of the prototype configuration system was how to separate control information (how to go about the configuration task, using heuristics if possible) from the object-level knowledge (ensuring that the configurations synthesized are always legal).

This system was implemented in around 5,500 lines of Quintus Prolog, and runs on a SUN workstation. It consists of an object-level knowledge base (components, attributes, *etc.*) together with heuristic knowledge, tactics, methods (specifications for tactics), and a planning mechanism to guide inference in the system. It was tested systematically: all tests were drawn up by myself in line with the training example available to me and customer specifications drawn up for other HP systems. Details may be found in Lowe (1993a).

We now explain the rationale behind the various design decisions that we took in designing the architecture for an automated configuration system.

6.2 Types

Adopting a hierarchical structure for storing knowledge seems initially attractive. However, in a field which is changing rapidly (as hardware is) this gives rise to considerable problems when attempting to fit new devices into a rigid framework. It is a common problem in Artificial Intelligence that initial classifications within frame-based systems and the like break down when new objects are introduced which defy the original classifications, or if the information is put to a different use. This leads to rethinking either the classification or the properties attached to slots or both; or else to messy exception-handling procedures. The problem in the computer hardware domain is that we cannot predict the course that technology will take. New products might cut across existing divisions: maintenance of the system would mean not simply updating the product data but also maintaining the structure. This would add an unnecessary overhead on to an already onerous task. Our aim is to make maintenance as straightforward as possible. The knowledge base part of the system can be updated by people who currently maintain product information — people who do not necessarily have the expertise needed to maintain a structure tree for the knowledge base; so that unless all future products conform to the existing structure the addition of just one "revolutionary" component will cause problems.

Our solution, therefore, is to adopt a fairly "flat" type system. Individual components (processors, memory modules, terminals, disk drives, tape drives, printers, channels, cables, card cages, *etc.*) are represented as atomic types. There are two ways in which this knowledge may be maintained:

1. New components of existing types may be added (or old ones deleted).
2. New types are added (and old ones, which have no members left, deleted).

The first is done by adding the component to the knowledge base: the name of the component, together with its type; and its attributes as appropriate (*e.g.* for a disk, this would include its capacity).

The second arises when a new *kind* of device is added. Inevitably there will be at least one member of the type. Each new component, together with its type, is added to the knowledge base in the normal way, but other information is also needed, such as how more complex terms may be built up using this type.

For example, suppose a new storage device is invented. Let us call this type of device **new_fangled_thing**.

Now suppose that *example* is an object of type **new_fangled_thing**. Then we add the information *example* : **new_fangled_thing** to the knowledge base. Suppose that all objects of type **new_fangled_thing** are to be connected via objects of type **channel**. Then we must assert that

$$\forall x : \mathbf{new\_fangled\_thing}. < x, y >: \mathbf{new\_fangled\_thing} \times \tau \rightarrow \tau = \mathbf{channel}$$

The problem of dealing with component *models* is conquered using function types. For example, suppose 7937H is a model of disk drive. Models are important because attributes properly belong with models, rather than with individual components. For example, all disk drives of this model have capacity 571Mb. We name the function 7937H which maps natural numbers to disks. In general, suppose *model_name* is a model of component of type τ. $\mathcal{N}$ is the type of positive natural numbers. Then we have the function

$$model\_name{:}\mathcal{N} \mapsto \tau$$

So, for example, instances of disk drives of this model in the configuration will be indexed by the natural numbers: 7937H(1), 7937H(2), ...

Other compound types used are list types (typically lists of objects needing to be grouped together in the configuration), and pairs, which we have already seen.

An object of type configuration is a member of a complex type, of the form

$$list(\mathbf{processor}) \times list(\mathbf{memory}) \times list(\mathbf{device})$$
$$\times list(\mathbf{connectors}) \times list(\mathbf{connections})$$

where a **cardcage-term** is a complex type, whose members represent a card cage and interface cards, and the connections between them. Other subterms, not explicitly shown above, represent, for example, the channels and the device-cable pairs connected to them.

Note that a simple interface would allow non-AI-expert people to add domain knowledge, which can then be validated.

6.3 Object-level rules

As an example, let us consider the rules concerning the fact that, in any configuration c, all interface cards must be configured in a card cage.

So the definition of a legal configuration c (or what it means for c to be a member of the type **configuration**) includes the conjunct

$$\forall ch{:}\mathbf{channel}.\ \exists cc{:}\mathbf{cardcage}.\ connected\_via(ch) = cc$$

In other words, for each and every interface ch, there is some card cage cc, such that ch is connected via cc.

Taking $x \in l$ to denote "x is a member of the list l", we say that ch is connected via cc in the configuration c **iff**

$$\begin{aligned} & cardcages(c) = ccs : list(\mathbf{cardcage}) \quad \wedge \ cc \in ccs \\ & \wedge \ cardcage\_terms(c) = ccts : list(\mathbf{cardcage} \times \mathbf{ic\text{-}lists}) \\ & \wedge \ < cc, \ldots, chtl, \ldots > \in ccts \quad \wedge \quad < ch, devices > \in chtl \end{aligned}$$

in other words, **iff** cc is a member of the list of cardcages of c, and $< cc, \ldots, chtl, \ldots >$ is a member of the list of cardcage-terms of c, and $< ch, devices >$ (the channel ch and its attendant devices) is a member of the channel-term $chtl$. Note that the type **ic-lists** is a complex type of cross-products of lists of cross products of interface cards and the components they connect. More examples may be found in Lowe (1991) and Lowe (1993a).

One constant worry with any system is how technological change will affect its maintainability. In extreme cases, it may be rendered redundant. The purpose of separating object-level from procedural and meta-level knowledge was to minimize, and localize, the changes which need to be made.

Clearly, a new component which is of an existing type should pose no problem. However, changes in technology mean that new types of component have to be added.

Completely new types are sometimes easier to deal with than types which seem to cut across existing type boundaries.

One example is that of data storage. We defined the types **disk** and **tape** type; however a newer product combines the functions of both these.

The following changes would need to be made to CLEM:

1. Add the new type, and the new component(s) to the knowledge base. Add acceptable cross-product types, so that legal connection of the device(s) are assured.
2. Allow the device type to appear in specifications, in the same way that CLEM allows $disks(c) = \ldots$, $tapes(c) = \ldots$ at present.
3. The current methods are adequate to deal with devices specified in this way (namely using $configure\_device\_list$, $configure\_device$) so no action is needed here.
4. Allow the *function* of these devices to be specified, *e.g.* $general\_storage(c) \geq \ldots$.
5. Write a new method for matching such a specification to appropriate devices, analogous to the method $total\_disk\_storage$.
6. Write a "higher-level" method which enables such a device to be synthesized in answer to less explicit needs. This is explained below.

With these additions to CLEM, requests for such devices can be made explicitly in specifications; and we can request a total storage capacity to fulfill all our storage, swap space and backup needs. However, in addition, if step 6 is taken, users can specify only primary user storage requirements, and have other needs (backup *etc.*) supplied automatically using the new device, as an alternative to the more traditional disk-tape combination.

We have not implemented such a higher level method in CLEM, as it needs expert knowledge relating to how storage needs are assessed or calculated, but such information is potentially available and could be used to update CLEM. Implementation is analogous to the present method for calculating backup needs.

The steps enumerated above are simple to implement compared with maintaining a more traditional rule-based system. The different levels involved can be advised by the different personnel affected: reflecting the division of labour between production, sales, engineering, and configuration experts.

6.4 Proofs

A typical specification is of the form

$$(processor(C) = ex\_proc) \quad \wedge \quad (capacity(disks(C)) \geq 4000) \quad \wedge \quad \ldots$$

A simple-minded, object-level proof proceeds as follows. The details have been omitted, and the proof simplified, in the interests of brevity. The full details of a simplified specification may be found in Lowe (1991).

1. The first conjunct is proved. In this case, this is the one which specifies the processor.

   The definition of the destructor function

   $$processor([Proc|\_]) = Proc$$

   is invoked by matching

   $$processor(C) = ex\_proc$$

   and C is partially instantiated to $[ex\_proc|\_]$.

2. The proof of the second conjunct proceeds as follows. We use the the definition of capacity and the destructor function $disks(C)$ to generate a list of disk drives and test that they are of the required capacity. When this has been done successfully, C will have been further instantiated.

3. The other conjuncts are proved in like manner.

4. When there are no other specification goals, we prove that C is in the type **configuration**. The proof of this synthesizes appropriate connections for the configuration object.

6.5 Tactics

A tactic is a program encapsulating a significant proof step with its attendant lower level steps. The latter are typically ones of less interest to the user of the system, and correspondingly harder to keep track of. An analogy with theorem proving would be the kind of well-formedness goals which are a feature of proof refinement systems such as Oyster (Horn, 1988) and Nuprl (Constable *et al*, 1986). We would prefer these to be taken care of automatically so that we can concentrate on the "interesting" proof steps.

So it is with configuration. For example, let us consider a Prolog tactic to configure a device, shown in Figure 1. The arguments of $configure\_device$ are the Device to be configured, an interface channel (IC), and the configuration (C) in which these occur.

```
configure_device(Device,IC,C):-
    connect_cable(Device,Cable,C),
    connected_via(Device,IC,C),
    type([Device,Cable,IC],_).
```

Figure 1: The tactic $configure\_device$

In order to configure a device, we need a cable for it, we need to connect the device *via* the cable to the interface, and the whole construction — the interface with the device-cable pair — must be well-formed. In Figure 1, *connect_cable*/3 ensures a Cable, *connected_via*/3 takes care of the connection *via* the interface, and *type*/2 is the well-formedness check.

Reasoning at the level of the tactic facilitates the search for an acceptable solution whilst ensuring that any such solutions found will be legal; *i.e.* they ensure the soundness of the automatic configuration system.

6.6 Methods

A method is a specification for a tactic. Figure 2 shows the general structure of a method.

Name	*method name*
input	*syntactic form of input goal*
output	*syntactic form of output goal*
preconditions	*for method to be applicable*
effects	*of applying tactic*
tactic	*program specified by method*

Figure 2: Method structure

Methods have slots for *method name*; *input*, which the input goal must match; *preconditions*, which are conditions which must be true of the input if the method is to be *applicable*; *output*, which will match the rewritten input goal if the method is applied; *effects*, which are conditions on this output goal if the method is applied; and the specified *tactic*: the program to be applied to the goal at this point.

Figure 3 gives the method which specifies the *configure_device* tactic we saw earlier.

$Name$	$configure\_device$
$input$	$configure(Device, IC, C)$
$output$	nil
$preconditions$	$Device\ is\ a\ device\ of\ C$ $and\ IC : \tau$ $and\ Device\ needs\ slot\ of\ type\ \tau$ $and\ the\ number\ of\ slots\ available\ of\ type\ \tau\ is\ s(n)$
$effects$	$the\ number\ of\ slots\ of\ type\ \tau\ available\ is\ n$
$tactic$	$configure\_device(Device, IC, C)$

Figure 3: The method $configure\_device$ (in informal English)

In this method, the goal to be proved must have the form $configure(Device, IC, C)$ and the output of the method is nil: this applies to all *terminating methods* where no further rewriting will be necessary if the tactic is successfully applied. The preconditions state, in order, that there is a Device to be configured in C, that IC is of type τ, $< Device, IC >$ inhabits a type, and that the number of slots available of the correct type is greater than zero. The effect of the method is to reduce the number of such slots by one.

There is a saving of effort in checking the method as opposed to trying the tactic. The latter must find a cable (or generate one) and actually make the allocation of device-cable pair to slot. Since the former is deterministic and not of great interest, and there are ways of guaranteeing the latter without actually doing it, we prefer to reason at the meta-, planning level, since the saving outweighs the overhead (see Section 6.8).

One nice feature of specifying tactics by methods in this way is that it models the "user" or "customer view" of configuration, as opposed to the "engineer view", which is "modelled" by actually executing the tactic. This makes developing good explanation facilities a realistic possibility. This is not true of rule or constraint based systems which work at a low level, where the search space is more complex and reasons for the choices made may not be readily apparent.

More importantly from the point of view of maintenance considerations, the use of methods gives an explicit place for us to write control information. For example, the fact that the tactic to configure an interface channel in a card cage should only be run if there are spare card cage slots belongs in the preconditions of the *configure_ic* method. The fact that a cardcage should not be configured if there are completely empty cardcages already present in the configuration finds its place in the *configure_cc* method.

6.7 Proof plans

The technique of proof planning is described in Bundy (1988). An account of how they were used to guide inference in inductive theorem proving may be found in Bundy *et al.* (1991).

In the context of configuration, we can think of proof plans as the expressions of meta-level strategies. The aim in proof planning is to find a plan tailor-made for the specification which will prove the particular theorem of the form of (1) introduced in Section 4 given to us. A sequence of applicable methods is found. If this sequence, which we call a *plan*, is executed, then every conjunct in the specification is proved and we are guaranteed that a well-formed configuration object meeting the specification will be instantiated as a by-product.

We can go further. High-level strategies for configuration can be developed, and encapsulated as proof plans, or supermethods. For example, let us consider the most basic strategy, which is:

1. Deciding an appropriate processor from information in the specification.
2. Each processor fixes a kind of "template" configuration: set this up *via* matching.
3. Attend to explicit user needs as given by the specification.
4. Add essential components not explicitly specified.

Within this strategy, we might also want to control the order of configuration of devices, as it can be shown that some sequences are (heuristically) better than others (Lowe, 1993a). Supermethods are distinguished from methods in that they call other methods (or supermethods) from within their effects slots. Some methods are iterators: for example, we may want to call a method to configure a device until there are no devices so far left unconfigured.

Within the overall guidance given by this plan, there is sufficient flexibility to cater for a variety of specifications. At the same time, the existence of a proof plan, which will be applied if possible, means that there is not a random choice of methods which could lead to legal but "unnatural" configurations being generated; moreover, backtracking in order to seek alternative solutions does not lead first merely to plans which contain the same methods, but applied in a different order — in other words, to trivially different solutions — but to configurations which are significantly different.

Various strategies have been developed: the constraint relaxation strategies referred to earlier; and configuring hardware to comply with cost guidelines, where the configuration objects referenced by methods are annotated so as to keep a running check on the approximate cost of the configuration: this cuts down on much unnecessary search, as branches leading to over-expensive non-solutions are pruned early from the search.

Again, this explicit representation of strategies means that we can have the benefit of efficiency whilst retaining a declarative, transparent system. The control knowledge does not have to be "hard-wired" deep in the program.

6.8 Testing the system

We have tested the prototype system CLEM (Lowe, 1993b), and experimented with it over a wide variety of specifications, using alternative formulation of methods, and by adding new strategies. In addition, we successfully added new component details, to test the maintainability of the object-level knowledge of the system.

CLEM and proved capable of handling all but very pathological specifications. It can successfully employ strategies for control and for taking design issues into account. We tested the maintainability of the system by adding a new processor and other components. This tested whether the object-level knowledge could be updated independently of the rest of the system. Maintainability of control information was tested throughout: control information tends to be learned gradually, and we were able to incorporate such understanding on an incremental basis, without needing to make major changes to the rest of the system.

Synthesizing a configuration in this system involves finding a plan, which is a sequence of methods, and then executing the plan, *i.e.* running the corresponding tactics in the sequence given by the plan. One problem with using planning, or any kind of meta-level reasoning, is the overhead incurred. For *very* simple specifications, leading to small systems, the overhead is not cancelled out by the saving in execution time, when we compare the total time of planning + execution with the time taken to find a solution using the object-level theory unaided. However, this is not the case when configuring large multi-user systems. In fact, the configurations do not have to be very large for planning to pay off (Lowe, 1993a). So one empirically proven advantage for this approach is that it is more efficient in finding the first solutions.

This is not the only advantage, however. One feature which was thought useful was the ability to search the planning space for alternative solutions, rather than having to execute each such solution before the next plan is found. In real life, the (human) configurer presents several solutions for the customer to choose from. The information provided at the planning level is sufficient for an informed choice to be made, since this level deals with components at the right level for the user: in terms of devices and attributes of whole configurations, rather than in terms of devices, cables, slots, *etc.* Only when an acceptable solution has been found at the planning level does this plan (and this only) need to be executed, to give the full details of the configuration.

Thus not only does the proof planning approach benefit system maintenance and ensure sound solutions, it also facilitates exploration of alternative solutions. It should also be said that testing the system was also helped by this "two-stage" approach in which most detailed information is presented only on execution, making easier to check the top-level details of the synthesized systems at the planning level first.

7 Conclusion

We have developed a formalization of the configuration domain in order to apply established theorem-proving techniques to the problem of configuring hardware to meet specifications. This ensures the underlying soundness of the system, in that all solutions generated will at least be legal objects. Using this as a basis, we have been able to encapsulate the higher-level reasoning, used informally by human experts, in a more formal way in order to facilitate the generation of well-designed and "natural" solutions, and to generate these solutions as efficiently as possible by guiding search by means of meta-level reasoning techniques. We would not argue that we have discovered, much less implemented, all strategic knowledge brought to bear on this problem by human experts. However, the methodology of this approach enables this to be done incrementally. This is due to the fact that the meta-level knowledge is captured independently of the object-level theory. So not only can we maintain the system by the addition of new components, and even new types of components, but we can also experiment with new strategies. Given that human expert strategies are often opaque at first, there is the added psychological benefit that we can gain new insights into the process of human reasoning in a complex task. Using the proof planning methodology, we can carry out these kinds of modifications and experiments in a principled way which is easy to track.

Any student of systems analysis knows that maintainability is an important criterion for judging the success of a computer system. Usually this question is assessed with regard to systems which have been in use, in the field, for some time. This clearly has not been done with CLEM, which is merely a prototype. However, CLEM was not born, complete and perfectly formed, but evolved over a period of many months. Its expandability was, therefore, an important issue right from the start, and not simply something to worry about for the future.

With CLEM, unlike many knowledge based systems, it is impossible to get over problems with quick "fixes": we claim that this is a benefit. The developer of the system is forced to make knowledge explicit and modular, and this can only be beneficial in the long run, in terms of saving programmer-hours.

8 Reflection

This special issue of IEEE addresses important questions which many believe to be long overdue in the field of AI. For too long questions of dependability, robustness, and maintenance (which are all interlinked) have been left, if not to chance, then at least in the background.

In this we can discern a kind of parody of the Kuhn view of paradigm (Kuhn, 1970): part of the attraction of a paradigm, once it becomes the "established" one, is that it leaves plenty of "holes". In science, this "hole-filling" activity can be seen as virtually all of what Kuhn calls "normal scientific activity". There is a good parallel in the field of expert systems — whether seen as "scientific activity" or not — in that maintaining XSEL, for example, is seen as a satisfying activity by the maintenance programmers who perform this task. In fact, it seems to be generally expected that rule-based systems will naturally require this kind of maintenance. This attitude is a dangerous one, however, if IKBS is to approach the dependability of other engineering disciplines (Bundy, 1987a). We believe such systems must be put on a firm logical basis as a first step towards this.

References

Bachant, Judith and Soloway, Elliot. (March 1989). The engineering of XCON. *Communications of the ACM*, 32(3):311–317.

Barker, Virginia E. and O'Connor, Dennis E. (March 1989). Expert systems for configuration at digital: XCON and beyond. *Communications of the ACM*, 32(3):298–310.

Bates, Joseph L. and Constable, Robert L. (January 1985). Proofs as programs. *ACM Transactions on Programming Languages and Systems*, 7(1):113–136.

Bundy, A. (1987a). AI bridges and dreams. Research Paper 300, Dept. of Artificial Intelligence, Edinburgh, Also in AI and Society, vol. 1 no. 1.

Bundy, A. (1987b). How to improve the reliability of expert systems. In Moralee, S., (ed.), *Research and Development in Expert Systems IV*, pages 3–17. C.U.P. Also available as Research Paper 336, Dept. of Artificial Intelligence, Edinburgh.

Bundy, A. (1988). The use of explicit plans to guide inductive proofs. In *9th Conference on Automated Deduction*, pages 111–120. Longer version available as Research Paper 349, Dept. of Artificial Intelligence, Edinburgh.

Bundy, A., van Harmelen, F., Hesketh, J. and Smaill, A. (1991). Experiments with proof plans for induction. *Journal of Automated Reasoning*, 7:303–324. Earlier version available as Research Paper 413, Dept. Of Artificial Intelligence, Edinburgh.

Constable, R.L., Allen, S.F., Bromley, H.M. *et al.* (1986). *Implementing Mathematics with the Nuprl Proof Development System.* Prentice Hall.

Frayman, F. and Mittal, S. (1987). Cossack: a constraints-based expert system for configuration tasks. In *Proceedings of the 2nd International Conference on Applications of AI to Engineering*, Boston MA.

Freeman, Michael W., (December 1985). Case study of the beacon project. IEEE Videoconference seminar on Expert Systems and Prolog.

Harmon, P. (1987). Digital equipment corp saves millions with expert systems! *Expert Systems Strategies*, 3(8):1–13.

Horn, C. (1988). The Nurprl proof development system. Working paper 214, Dept. of Artificial Intelligence, Edinburgh, The Edinburgh version of Nurprl has been renamed Oyster.

Kuhn, Thomas. (1970). *The structure of scientific revolutions.* University of Chicago Press, second edition.

Lowe, Helen. (1991). The use of theorem proving techniques in expert systems for configuration. In Rault, J.-C., (ed.), *Proceedings of the Eleventh International Workshop on Expert Systems and their Applications, Avignon.* EC2. Also available from Edinburgh as Research Paper 536.

Lowe, Helen. (1993a). *The application of proof plans to computer configuration problems.* Unpublished Ph.D. thesis, University of Edinburgh.

Lowe, Helen. (1993b). The CLEM configuration system, user manual and programmer manual. Technical Paper (forthcoming), Dept. of Artificial Intelligence, Edinburgh.

McDermott, John. (1982). R1: a rule-based configurer of computer systems. *Artificial Intelligence*, 19:39–88.

Mumford, Enid and Macdonald, W.B. (1989). *XSEL's progress: the continuing journey of an expert system.* Chichester Wiley.

Rosenbloom, Paul S., Laird, John E., McDermott, John, Newell, Allen and Orciuch, Edmund. (September 1985). R1-soar: an experiment in knowledge-intensive programming in a problem-solving architecture. *IEEE Transactions on Pattern Analysis and machine intelligence*, PAMI-7(5):561–568.

Searls, David B. and Norton, Lewis M. (1990). Logic-based configuration with a semantic network. *Journal of Logic Programming*, 1:53–73.

Soloway, E., Bachant, J. and Jensen, K. (July 1987). Assessing the maintainability of XCON-in-RIME: coping with the problems of a very large rule-base. In *6th National Conference on Artificial Intelligence*, Philadelphia. Sponsered by the American Association for Artificial Intelligence.